# Lecture 1 - Introduction

*"Give someone a program, you frustrate them for a day; teach them how to program, you frustrate them for a lifetime." - David Leinweber*

- Module Overview
- Content
- Assessment

## About

- Background: CS, SE and Game Dev
- Researcher in Automated Diagram Generation
- Shipped code in C++, Java, Kotlin & others
- Author and maintainer of FXGL game engine
- YouTube game dev channel

## Structure

- 2hrs lecture + 2hrs tutorials per week
- "C++" is just a tool, "game engine development" is theory
- As a result, you will be able to understand / use any game engine
- Unlike previous modules, this one covers low-level concepts and their high-level applications

## Expectations (Year 1)

minecraft

+++

## Reality (Year 1)

minecraft

## Expectations (Year 2)

minecraft

+++

## Reality (Year 2)

minecraft

## Expectations (Year 3)

minecraft

+++

## Reality (Year 3)

minecraft

## Content - Language & API

- C++ 14+
- SDL 2 (users include Valve and Unreal Engine)

Beware online resources/tutorials that use old C++ or SDL 1

## Assumptions

- You know: Variables, Conditionals and Iteration
- Used to some extent: Functions, Classes and Memory Management
- We will build on your knowledge from other modules

## Toolchain

- Windows + MS Visual Studio 2017+ (as agreed)
- (optional but strongly recommended) GitHub + git

## Content Overview

Main topics + games / tech that act as case studies for the topics.

## Game Engine Architecture (Unreal, Unity)

- how similar is a game engine architecture to a game architecture?
- what subsystems exist and what do they do?

## Application Framework (Godot, FXGL)

- events and event bus
- how do these subsystems communicate with each other?
- how do they communicate with the game?

## Entity-Component System (Overwatch, Minecraft)

- how do we represent game objects?
- where do we store game objects?
- how do we manage many game objects?

## Physics subsystem (Arx Fatalis, Sony PS3, Valve games)

- basic vector maths
- simulation and collision detection
- how do game objects interact with each other?
- how do they follow physics rules?
- what approximations do we apply to game objects simulation?

## Graphics subsystem (Star Wars Jedi Academy, Metal Gear Solid 5)

- how do game objects get drawn to the screen?
- how do we talk to the GPU?
- how can we abstract away the low-level complex details to simplify our representation of drawing to the screen?

## Audio subsystem (Telltale games)

- how do we store sound effects and music?
- how can we play these stored audio files?
- how can we implement positional sound?

## AI subsystem (F.E.A.R, Assassin's Creed)

- how do game objects know what and when to do?
- how to navigate the game world without passing through obstacles?
- what types of AI are most common in modern games?

## Scripting and domain-specific languages (The Elder Scrolls V: Skyrim)

- how can we write bespoke code per game object without adding complexity to engine/game code?
- how can we improve the compile time of game code?
- how can we make it easy for the community to contribute their additions to the game without changing any source code?

## Achievements / gameplay (Dragon Age Inquisition)

- how can we integrate an achievement system into an engine?
- how can we implement in-game tutorials, side quests, quick-time events, cutscenes, dialogues and other gameplay related features?

## External tool integration (Tiled Map Editor, 3ds Max)

- how do we benefit from supporting external tools?
- how can we add support for external tools?

## Support

- Lectures only provide the theory basics (you shouldn't rely just on lecture material)
- Tutorials only provide the practice basics (you shouldn't rely just on tutorial material)
- Video material / tutorials

Strongly recommended: - Game Engine Architecture (both hard copy and e-book are available from the library). - Game progamming patterns - C++ - SDL 2 - Anything (reasonably up to date) you can find online

## Assessment

- 100% of the module mark
- Details on StudentCentral

Expectations from you at L5 are higher than at L4. Attendance is important. 4-6 hours of independent study each week.

## Demo time

Our own tiny engine.

- XCube2D

## C++ Basics

- Compile & link & run a C++ application
- Variables, Conditionals, Iteration, Printing (terminal)

## Compile

Compiling takes C++ source files and produces an 'object' file.
(Nothing to do with the term 'object' in object-oriented software).

```
$ g++ -c Main.cpp
```

## Link

Linking combines ‘object’ files into an executable.

```
$ g++ -o Main Main.o
```

## Run

- Double-click on `Main` (`Main.exe` on Windows) in the graphical file manager, or
- `cd` to the containing directory and execute ‘./Main

## Pragmatic Approach to C++

We will pretend some things do no exist or some things are simpler than they really are. This approach will help us progress quicker.

## Questions

You should know this from previous modules. Let's recap.

What is ...

- a variable?
- a data type?
- a conditional statement?
- an iteration?

## Recall Java Primitive Data Types

So far, you heavily focused on Java. Let's recap and see how C++ differs.

- Hint 1: there are 8 of them. |
- Hint 2: String is NOT one of them! |
- Hint 3: they start with a lowercase letter. |

## Common C++ vs Java Data Types

- `int` for whole numbers
- `double` for real numbers (e.g. with a decimal point)
- `bool` for Java `boolean` (careful! some old code may use `int` as `bool`)
- `char` for characters (careful with non-English characters)
- `std::string` for literal text (like Java `String`)

## Variables

Declaration and assignment:

```
int x;
x = 7;
```

or

```
int x = 7;
```

## Conditionals

```
if (x <= 0) {
    // x less than or equal to 0
}
```

## Iteration

```
for (int i = 0; i < 10; i++) {
    // runs 10 times
}
```

## Printing to Terminal

Note the `endl;` at the end.

```
std::cout << "Hello World" << std::endl;
```

## Sample Code

```
for (int i = 0; i < 10; i++) {
    if (i < 5) {
        std::cout << "i is less than 5. i is " << i << std
    }
}
```

## Solve the Problem in C++

Print the following:

```
1 * 10 = 10
3 * 10 = 30
5 * 10 = 50
7 * 10 = 70
9 * 10 = 90
```

## Functions

A program can be divided into small chunks. These chunks of code are called **functions** (in Java they are called methods):

```
int add(int x, int y) {
    return x + y;
}
```

Note: - how to pass variables in and out of a function, and - the scope of variables in a function.

## Functions (example)

Isolate some code and forget about it.

```
void println(std::string text) {
    std::cout << text << std::endl;
}
```

1. Can we add functions to our previous solution?
2. Will it help to make the code more readable?
3. Why yes / no? Any difference if it was a larger piece of code?

## Conclusion

- Problem-solving is the same as in (or similar to) Java
- Basic C++ syntax is straightforward

## Next Week

- C++ Basics (Cont.)

## Tutorial

Tutorial link